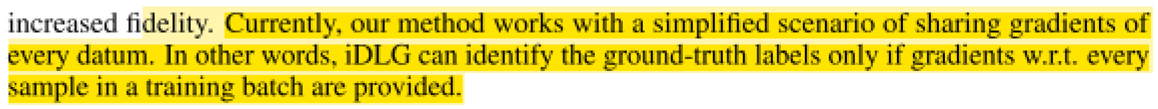
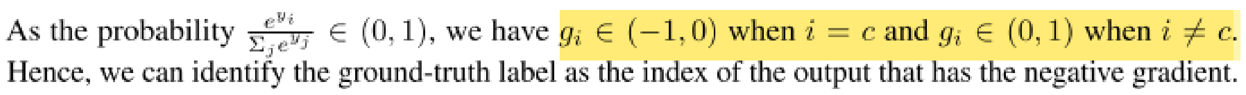
<u>**Status Update 14.07.20**</u>

The authors from iDLG did a good improvement for DLG by including label prediction from gradients in their recreation setup. Their prediction method works 100% in batch size 1 scenarios. For other batchsizes it is not defined. Instead they changed the scenario and said the victim should just send them the gradients from each individual sample of a batch, just lika in a batch size 1 scenario.


increased fidelity. Currently, our method works with a simplified scenario of sharing gradients of every datum. In other words, iDLG can identify the ground-truth labels only if gradients w.r.t. every sample in a training batch are provided.

I try to develop a prediction strategie, that might not reach the 100% prediction rate, but that doesn't cheat the scenario like they did.

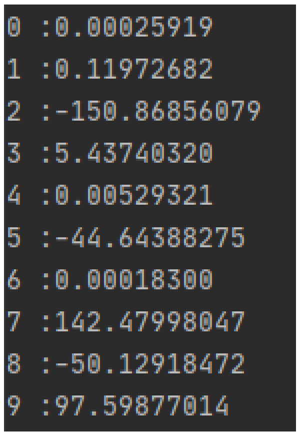
Their approach is to take the second last layer of gradients and find the negative value, which they logg in as their prediction candidate.


As the probability $\frac{e^{y_i}}{\Sigma_j e^{y_j}} \in (0,1)$, we have $g_i \in (-1,0)$ when $i = c$ and $g_i \in (0,1)$ when $i \neq c$. Hence, we can identify the ground-truth label as the index of the output that has the negative gradient.

Starting with a randomly generated sample they alter it in order to optimize the result. To archieve this, they minimize the difference between the originally attacked gradients and the gradients, the network produces when the dummy is inserted and the weights get fitted toward the predicted output.

For batch sizes larger than 1 the predicted gradient values start to overlay each other. However it is still possible to predict the labels. Even though negative values might become positive in this overlayed state and the result gets more blurry. Negative values hint the classes and the magnitude of the values hints how often a value is present in the batch.

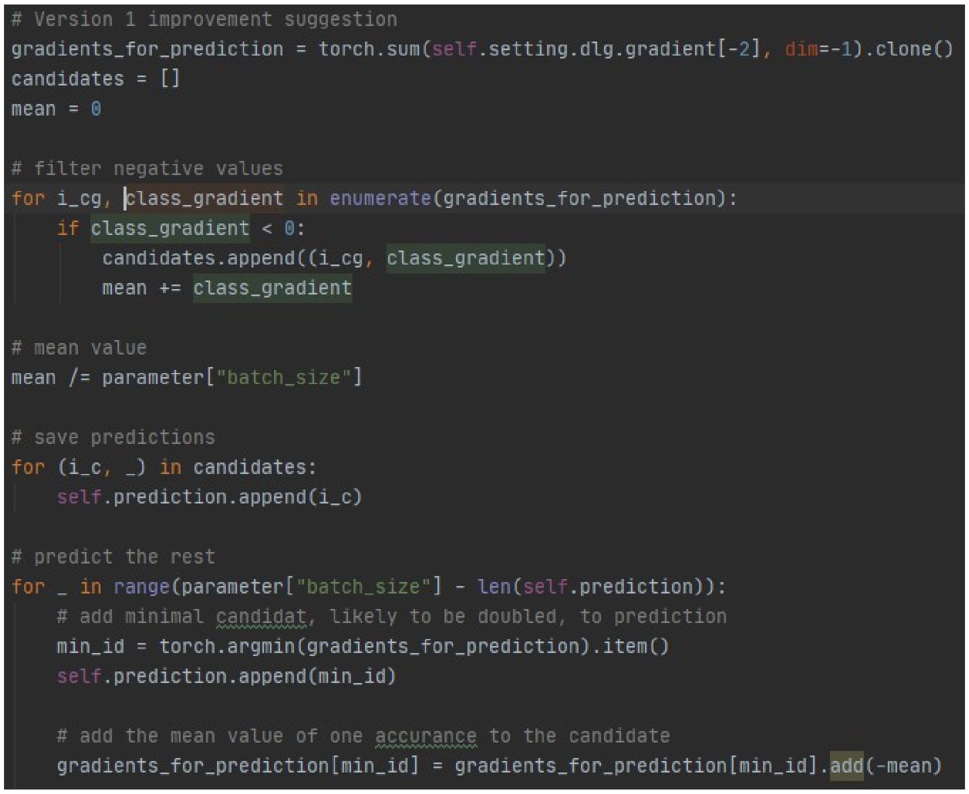
For example: the following classes [2, 2, 2, 5, 7, 8] produced this batch


```
0 :0.00025919
1 :0.11972682
2 :-150.86856079
3 :5.43740320
4 :0.00529321
5 :-44.64388275
6 :0.00018300
7 :142.47998047
8 :-50.12918472
9 :97.59877014
```

My first draft for a prediction strategie will proceed in two steps. Step one is to take every negative value. These are pretty likely to be present in the original batch. In this case [2, 5, 8]. That means 5 more values to predict. In order to get a feeling for the impact one occurance of a class makes, I take the negative values add them together and devide the result by the batchsize. In this case about -40.

So fore the next prediction step I choose the smallest value, add it to my prediction and reduce its value by the mean value unitl my prediction list reaches the size of the batch.

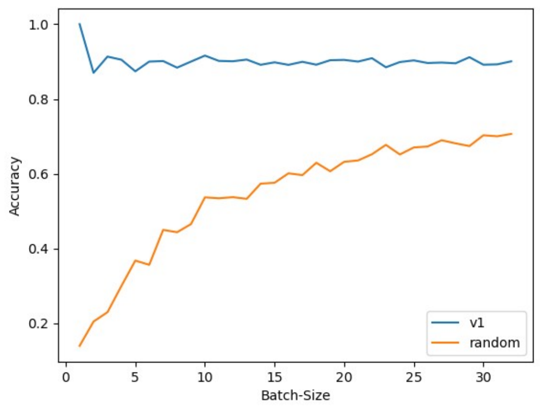
Code for my prediction:



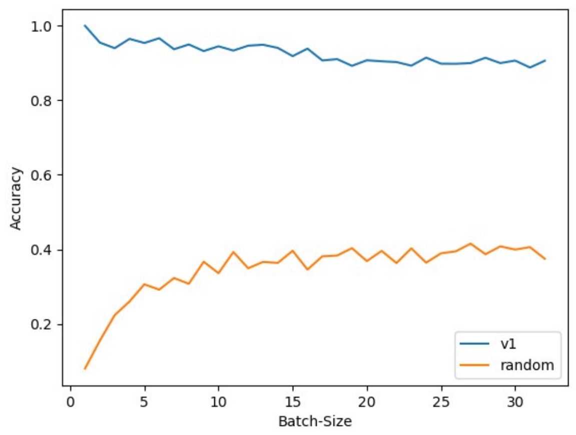
```
# Version 1 improvement suggestion
gradients_for_prediction = torch.sum(self.setting.dlg.gradient[-2], dim=-1).clone()
candidates = []
mean = 0

# filter negative values
for i_cg, class_gradient in enumerate(gradients_for_prediction):
    if class_gradient < 0:
        candidates.append((i_cg, class_gradient))
        mean += class_gradient

# mean value
mean /= parameter["batch_size"]

# save predictions
for (i_c, _) in candidates:
    self.prediction.append(i_c)

# predict the rest
for _ in range(parameter["batch_size"] - len(self.prediction)):
    # add minimal candidat, likely to be doubled, to prediction
    min_id = torch.argmin(gradients_for_prediction).item()
    self.prediction.append(min_id)

    # add the mean value of one accurance to the candidate
    gradients_for_prediction[min_id] = gradients_for_prediction[min_id].add(-mean)
```

In my example the algorithm suggests:
[2, 2, 2, 2, 5, 8,]
instead of the original
[2, 2, 2, 5, 7, 8]

5/6 Accuracy is pretty decent. As the following graph suggests, this holds for larger batch sizes. Here you can see a graph off accuracy versus batchsizes [1, 32]. average of 100 runs .



IID IID Data



Biased Data: 50% [0], 25% [1], 25% random [2-9]

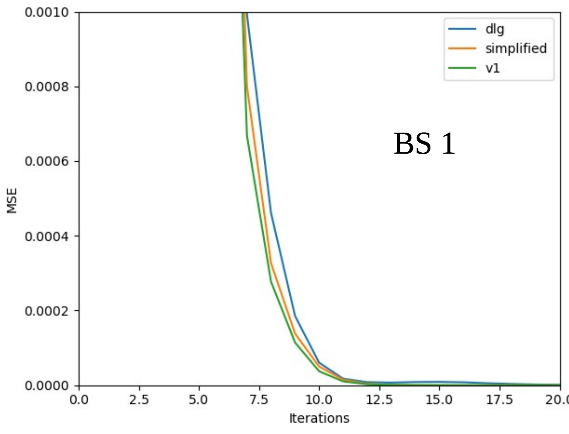
However, as you can see, the class 7 occurrence in the original batch, for some reason caused a huge positive value. There is for sure room for improvement of this strategie. I currently have the following ideas:

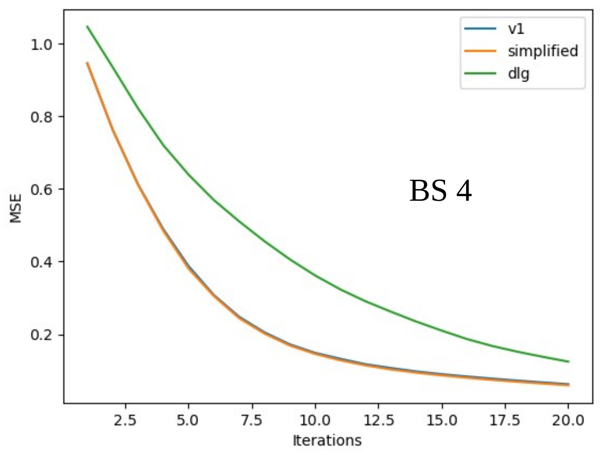
- Include absolute value of the gradients
- Create multiple fake batches first, so that the attacker can calculate the mean value from that more precisely.
- Create multiple fake batches, train a neural network to predict classes from the fakes and then use the network for our prediction task

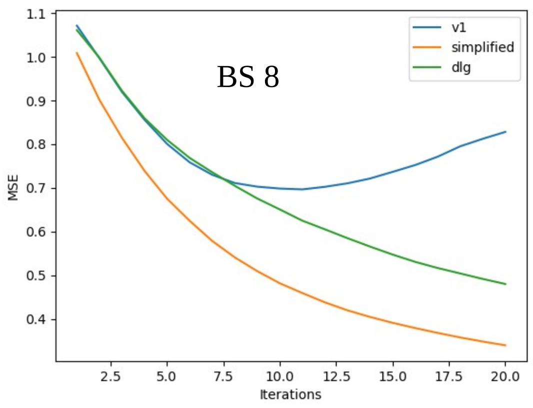
But before I work more on these additional strategies I have to evaluate the reconstructed images first, because this is the whole point of the task anyway.
My 90% prediction + iDLG won't be as good as the 100% (cheated) prediction + iDLG from the paper. The interesting part is, whether it can beat the traditional DLG without bending the scenario.

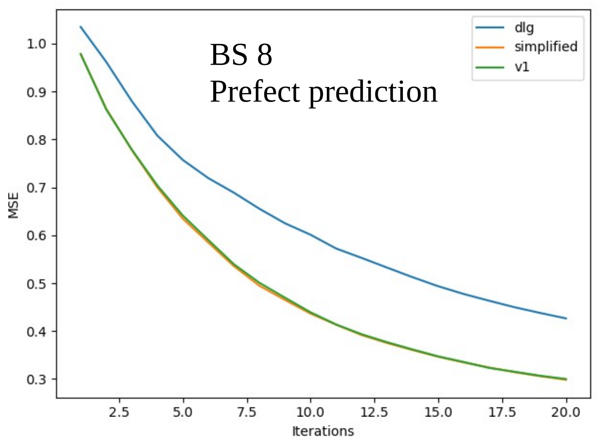
On my machine it takes a very long time to run the image recreations. For a comprehensive study I'd have to run the recreation for different batch sizes and for different strategies several time to get average values while I adapt my strategies. Maybe we can get on a computer cluster for this task.

I did some test runs:
For bs = 1 there is almost no difference between the strategies
For small batchsizes bs=4 my algorithm has as good performance as the idlg, which is better than dlg.
For larger batchsizes bs =8 my algorithm starts to make prediction mistakes, which result in higher recreation error. However the runs where the prediction went perfect let to good results

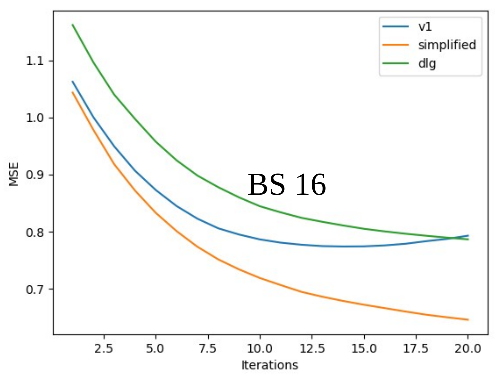


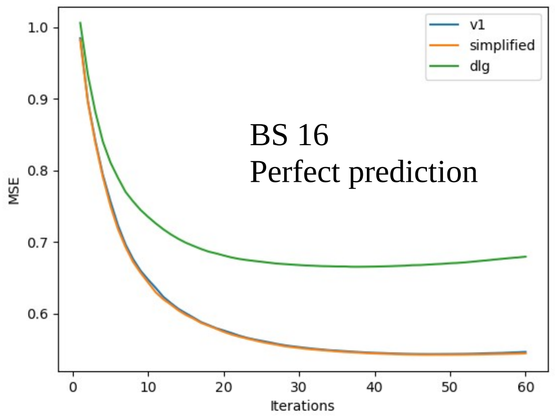




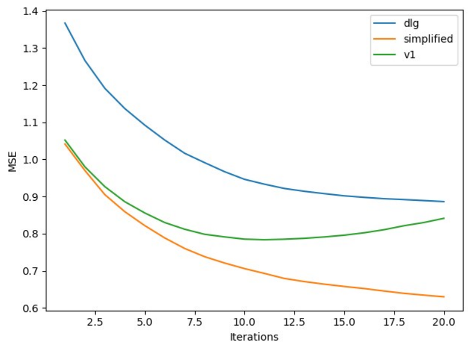


I got the same results at bs16



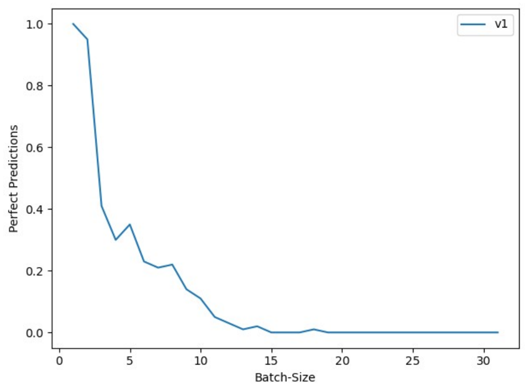


In this run the prediction got 1/16 labels wrong and did not converge.



So we can see even 1 mistake makes v1 perform worse.

One interesting aspect might be to have a look on how likely it is to get a perfect prediction. Unfortunatly this is worse than I expected.


v1
Perfect Predictions
Batch-Size